CS 410 / 510
Mastery in Programming

Chapter 12
MESI Protocol for MP Cache Coherence

Herbert G. Mayer, PSU CS
Status 8/13/2012

Cache Coherence in MP Systems (11/29/2011)

We discuss shared-memory MP systems, in which each processor has a *second level* (or L2) cache in addition to an internal *first level* (or L1) cache. Many discussions are specific to the cache implementation on the Intel® Pentium® processor family, with focus on the *MESI* protocol. Reference point is a text by MindShare Inc., literature reference [1]. The 7 case studies have been taken from [1]. *MESI* is used on the Pentium Pro processor family in a modified way.

The Cache Coherence Problem

The problem addressed and solved with the *MESI* protocol is the *coherence of memory and caches* on shared-memory MP systems. Even in a UP system, in which the processor has an internal (data) cache, memory and cache must have coherent data, or, if not, must have a mechanism to ensure that at critical moments the diverging copies will match again. Staleness of memory should be short-term and must be handled safely.

On shared-memory MP architectures this problem of data consistency (cache *coherenc*e) is magnified by the number of processors sharing memory, and by the fact that without an additional L2 cache performance would drop noticeably. In an MP system with N processors and 2 levels of cache there can be up to N*2+1 copies of the same data; the +1 stemming from the original copy of data in memory. These copies must all be *coherent.*

Synopsis

Definitions

See also the definitions in the introductory chapter on *caches*.

Allocate-on-Write:

If a store instruction experiences a cache miss, and as a result a cache line is filled, then the *allocate-on-write* cache policy is used. If the write miss causes the paragraph from memory to be streamed into a data cache line, we say the cache uses *allocate-on-write*. Pentium processors, for example, do not use *allocate-on-write*. Antonym: *write by*.

Back-off:

If processor P_1 issues a store to a data address shared with another processor P_2, and P_2 has cached and modified the same data, a chance for data inconsistency arises. To avoid this, the cache with the modified P_2 line must *snoop* for all accesses, read or write, to guarantee delivery of the newest data. Once the snoop detects the access request from P_1, P_1 must be prevented to get ownership of the data; this is accomplished by temporarily preventing access to the system bus. This denial for the purpose of preserving data integrity is called *back-off*.

Blocking Cache:

Let a *cache miss* result in streaming-in of a line. If during that stream-in no more accesses can be made to this cache until the data transfer is complete, then this cache is called *blocking*. Antonym: *non-blocking*. Generally, a *blocking cache* yields lower performance than a *non-blocking*.

Bus Master:

Only one of the devices connected to a system bus has the right to send signals onto the bus; this ownership is called being the *bus master*. Initially the Memory and IO Controller (*MIOC*) is the bus master; it also is possible a chipset includes a special-purpose bus arbiter. Over time, all processors, and for the processors their caches, request to become *bus master* for some number of bus cycles. The *MIOC* can grant this right, yet each of the processors (more specifically: its cache) can request a *back-off*, even if otherwise the right to be bus master would be granted.

Directory:

The collection of all *tags* is referred to as the cache *directory.* Note that in addition to the directory and the actual data there may be further overhead bits in a data cache.

Dirty Bit:

State bit associated with a cache line. This bit expresses whether a write hit has occurred on a system applying *write back*. Synonym: *Modified bit*.

Invalid:

State in the *MESI* protocol. This *I state* (possibly implemented via special purpose bit) indicates that the associated cache line is *invalid*, and consequently holds no valid data. This *invalid (I) state* is always set after a system reset.

Exclusive:

State in the *MESI* protocol. The *E state* indicates that the current cache is not aware of any other cache sharing the same information, and that the line is unmodified. It allows that in the future another line will contain the same information, in which case the *E state* must be changed. Also it is possible that a higher-level cache (L1 for example viewed from an L2) may actually have a shared copy of the line in exclusive state; however that level of sharing is transparent to other potentially sharing agents outside the current processor.

Paragraph:

Conceptual, aligned, fixed-size area of the logical address space that can be loaded completely into the cache. The holding area in the cache of paragraph-size is called a *line*. In addition to the actual *data*, a line in cache has further information, including the *dirty* and *valid* bit (in UP systems), the *tag*, *LRU* information, and in MP systems the *MESI* bits. The *MESI* I state and the *valid* bit in UP architectures perform the same function. Also, the *MESI* M state corresponds to the dirty bit in a UP system.

MESI:

Acronym for *Modified, Exclusive, Shared* and *Invalid*. This is a protocol to ensure cache coherence on the Pentium processor. A protocol is necessary, since more than one processor has a copy of common data with the right to modify. Through the *MESI* protocol data *coherence* is ensured no matter which of the processors performs writes. AKA as *Illinois protocol* due to its origin at the University of Illinois at Urbana-Champaign.

Modified:

State in the *MESI* protocol. The *M state* implies that the cache line found by a write hit was *exclusive*, and that the current processor has modified the data. The *modified* state expresses: Currently not *shared, exclusively* owned data have been modified. In a UP system, this is generally expressed by the *dirty* bit.

Shared:

State in the *MESI* protocol. The *S state* expresses that the hit line is present in more than one cache. Moreover, the current cache (with the *shared state*) has not modified the line after stream-in. Another cache of the same processor may be such a sharing agent. For example, in a two level cache,

the L2 cache will hold all data present in the L1 cache. Similarly, another processor's L2 cache may share data with the current processor's L2 cache.

Snarfing:

Scenario 1: ***c*** has a modified line and ***a*** wants to read:

A *snooping* cache ***c*** detects that another bus agent ***a*** wants to read a paragraph into ***a***'s cache line, of which ***c*** has a modified **M** copy. **M** of the MESI protocol implies exclusivity; no other cache will have a copy at this time. Instead of ***c*** 1.) causing ***a*** to back-off, 2.) then ***c*** streaming-out the line, 3.) ***a*** streaming-in that written paragraph, and 4.) both ***a*** and ***c*** ending up in ***S*** state, *snarfing* does the following: ***c*** streams-out the line and switches to **S**, but does not cause ***a*** to back-off. Instead, ***a*** reads the line from the bus during the stream-out process. This saves a full memory access and saves the back-off delay.

Scenario 2: ***c*** has a modified line and ***a*** wants to write:

A *snooping* cache ***c*** detects that another bus agent ***a*** wants to stream-in a paragraph due to *allocate-on-write*, of which ***c*** has a modified **M** copy. Note: ***a*** does not have a copy, but uses allocate-on-write, hence makes it known that the line will be streamed-in and then modified once present. Instead of ***c*** causing 1.) ***a*** to back-off, 2.) then streaming-out the line, 3.) switching to **I** invalid, and 4.) letting ***a*** stream-in the line and modify it, *snarfing* does the following: ***c*** streams-out the line, switches to **I**, but does not cause ***a*** to back-off. Instead, ***a*** reads the line directly from the bus during the stream-out process. This saves a complete memory access, and saves the back-off delay. Now ***a*** has the modified copy (modified by ***c***), as does memory, and **E** is the proper state for ***a***. Now ***a*** can further modify the line, resulting in a state transition from **E** to **M**. ***c*** no longer holds the line.

Snooping:

After a *line* write-hit in a cache using *write back*, the data in cache and memory are no longer identical. In accordance with the *write back* policy, memory will be written eventually, but until then memory is *stale*. The modifier (the cache that wrote) must pay attention to other bus masters trying to access the same line. If this is detected, action must be taken to ensure data integrity. This *paying attention* is called *snooping*. The right action may be forcing a back-off, or *snarfing*, or yet something else that ensures data coherence. *Snooping* starts with the lowest-order cache, here the L2 cache. If appropriate, L2 lets L1 *snoop* for the same address, because L1 may have further modified the line.

Squashing:

In a *non-blocking* cache, a subsequent memory access may be issued even if a previous miss resulted in a slow stream-in to the addressed cache line. The

subsequent memory access will be a miss again, which is being queued. Whenever an access references an address for which a request is already outstanding, the duplicate request to stream-in can be skipped. Not entering this in the queue is called *squashing*. The second and any further outstanding memory access can be resolved, once the first stream-in results in the line being present in the cache.

Strong Write Order:

A policy ensuring that memory writes occur in the same order as the store operations in the executing object code. Antonym: *Weak order*. The advantage of *weak ordering* can be speed gain, allowing a compiler or cache policy to schedule instructions out of order; this requires some other policy to ensure data integrity.

Stream-In:

The movement of a paragraph from memory into a cache line. Since line length generally exceeds the bus width (i.e. exceeds the number of bytes that can be move in a single bus transaction), a stream-in process requires multiple bus transactions in a row. It is thus possible that the byte actually needed arrives last in a cache line during a sequence of bus transactions. Antonym: Stream-out.

Stream-Out:

The movement of one line of modified data from cache into a memory paragraph. Antonym: Stream-in.

Weak Write Order:

A policy allowing (a compiler or cache) that memory writes may occur in a different order than their associated store operations. Antonym: *Strong Write Order*. The advantage of weak ordering is potential speed gain.

Write Back:

Cache write policy that keeps a line of data (a paragraph) in the cache even after a write. The changed state must be remembered via the *dirty* bit. Upon retirement, any dirty line must be copied back into memory. Advantage: only one stream-out, no matter how many write hits did occur to that same line.

Write By:

Cache write policy, in which the cache is not accessed on a write miss, even if there are cache lines in **I** state. A cache using *write by* "hopes" that soon there may be a load, which will result in a miss and then stream-in the appropriate line. And if not, it was not necessary to stream-in the line in the first place. Antonym: *allocate-on-write*.

Write Once:

Cache write policy that starts out as *write through* and changes to *write back* after the first write hit to a line. This is a typical policy imposed onto a higher level L1 cache by the L2 cache. Advantage: The L1 cache places no unnecessary traffic onto the system bus upon a cache-write hit. The lower level L2 cache can remember that a write has occurred by setting the *MESI* state to *modified*.

Write Through:

Cache write policy that writes data to memory upon a write hit. Thus, cache and main memory are always in synch. Disadvantage: repeated memory access traffic on the bus.

Write Once Policy and the MESI Protocol

The *MESI* protocol is one implementation of enforcing data integrity among caches sharing data; the *write once* write policy is a method to keep the protocol performing efficiently by avoiding superfluous data traffic on the system bus. First we'll discuss *write once*, then the MESI protocol.

Write Once Policy

Write through has the advantage of keeping cache and memory continually consistent. Draw-back is the added traffic placed on the system bus. *Write back* has the advantage of postponing unnecessary bus traffic until the last possible moment and to do so just once, even if many writes to a shared cache line occurred. The draw-back is the temporary inconsistency between cache line and memory. To avoid catastrophe, a *dirty bit* must mark the fact that at least one write happened to an exclusively owned line. *Write once* combines the advantages of both. For efficiency, multi-level caches generally use *write back* write for L2. *Write once* is a refinement used for L1 caches. Both use the *MESI* protocol to preserve data consistency.

In *write once*, L1 starts out using *write through*, and any line shared with an L2 cache is marked S, for *shared*. The corresponding copy in L2 is also marked S. If a write hit occurs, the modified data are written through to the L2 cache, which in turn also marks its line as modified, M. This transition is used by L2 to cause L1's write policy to change from *write through* to *write back*. Also, the L1 line is marked E, for *exclusive*. This **may look strange**, but is safe, since the M information is recorded in L2, the first to initiate snooping.

Subsequently, when the same processor modifies the same data further, the L1 cache experiences a write hit. This time, however, the L1 cache is in *write back* mode, and changes from E to M. L1 does not change the L2 cache again. Further writes keep the state in M. Of the two lines with the same paragraph addresses, the one in the L1 cache is more current than the one in L2. Both record the M state.

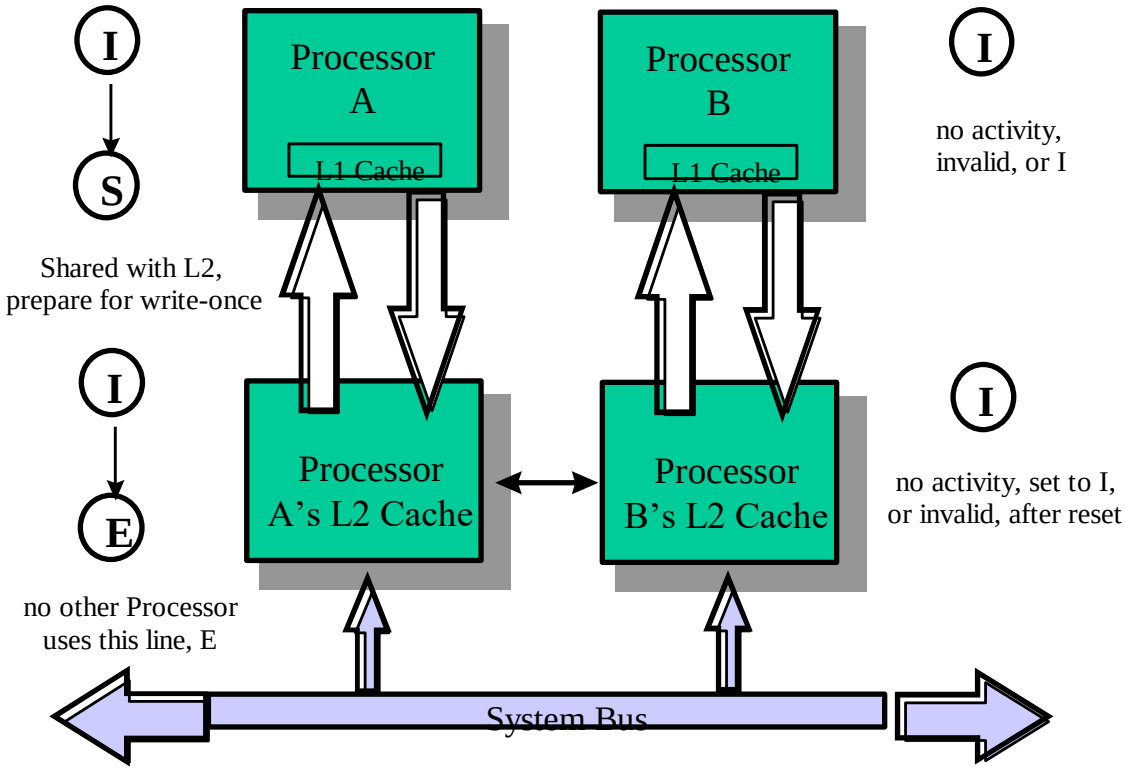
When another processor issues a read from the same paragraph address, the L2 cache with a modified copy of that line snoops and asks the other to back-off. As a result, the line will be written into memory. First, however, L2 must check if L1 has modified the data as well. This is detectable by the M state of L1 and the data are first flushed from L1 to L2, and then to memory. Finally, both L1 and L2 change to S. Otherwise, if the L1 cache has not further modified the data, indicated by E, L1 and L2 are already in synch, only the L2 line needs to be written to memory, and both L1 and L2 transition to S.

MESI Protocol

On a Pentium, each cache line may have its own write policy, independent of other lines even in the same set. The complete, total state of a cache line therefore, is expressed in the write policy used and the MESI state bits, associated with each line. These bits are: M for *modified*, E for *exclusive*, S for *shared*, and I for *invalid*. Initially a line holds no information, so its state is I.

During a system reset, the *MESI* bits for the Pentium's L1 and L2 caches are set to I. This will mark all lines as empty and will force any cache read or write access to *miss*. At the first read, lines will be streamed into L2 and a portion of those to be streamed into L1. Since L1 has a copy of what is in L2, L1 will be set to S and L2 to E. E holds, as long as no other processor's cache shares the same data. This transition is shown in the figure below. Note that below the other processor B has not yet made any data accesses, hence its cache lines remain I.

Initial **Read** of cache line into A after reset, B inactive



The table below explains the meaning of the 4 *MESI* states. Note that both caches have *MESI* bits, and both use *write back* most of the time on the Pentium family.

Table of MESI State Bits

MESI State	Description
Modified	The line has been changed by the cache; a store has taken place. It is implied that the data are *exclusive.* M alerts the cache to take action on a snoop hit. In that case the line is written back and the state is adjusted, generally to *Shared*. Alternatively, the cache can *snarf*. Done on Pentium ® Pro, not on Pentium.
Exclusive	The line is owned by just this processor. Except for memory, a *shared* copy may only exist in a higher order cache of the same processor. For example, L2 may label a line E that exists also in its L1. But no other processor's cache holds a copy of this same line.
Shared	The line is present in at least one other cache, maybe in several. However, all of these are identical copies. No other line with these data has performed a write. Shared implies unmodified.
Invalid	The line is not valid in this cache. Typical state after system reset, and thus the line is ready for receiving new data.

Every cache line will be in one of these 4 states. The state is influenced by the owning processors' action (load and stores) or by a bus snoop, when another processor addresses the same line. The latter is supported by special pins and connections (lines) between L2 caches on the Pentium processor. These lines are HIT# (for: cache hit) and HITM# (for: cache hit modified).

The Life and Fate of a Cache Line: 7 MESI Scenarios

This section shows typical state transitions of the L1 and L2 caches, each transition characterized by the respective title. A bulleted list explains the initial state, the figure shows the transition, and a trailing bulleted list highlights the key points as a result of the transition.

First we show 3 situations, in which processor B wishes to read a line of which processor A has a copy. Processor A has performed 0, 1, or more writes to that line.

Table of 3 Scenarios: B Reads some Line That is Also in A

Scenario	Processor A Status	A L1 State	A L2 State
1	A holds line from memory but has not *modified* it.	S	E
2	A has written a line once, using *write through.*	E	M
3	A has written the same line more than once, uses *write back* write policy after first write.	M	M

Then we show 4 situations, in which processor B writes to a line of which A has a copy. Again, A has modified the line various times, and B has read the shared data in one case before it attempts the write. We assume here the caches use write-by, NOT allocate-on-write.

Table of 4 more Scenarios: B Writes some Line That is Also in A

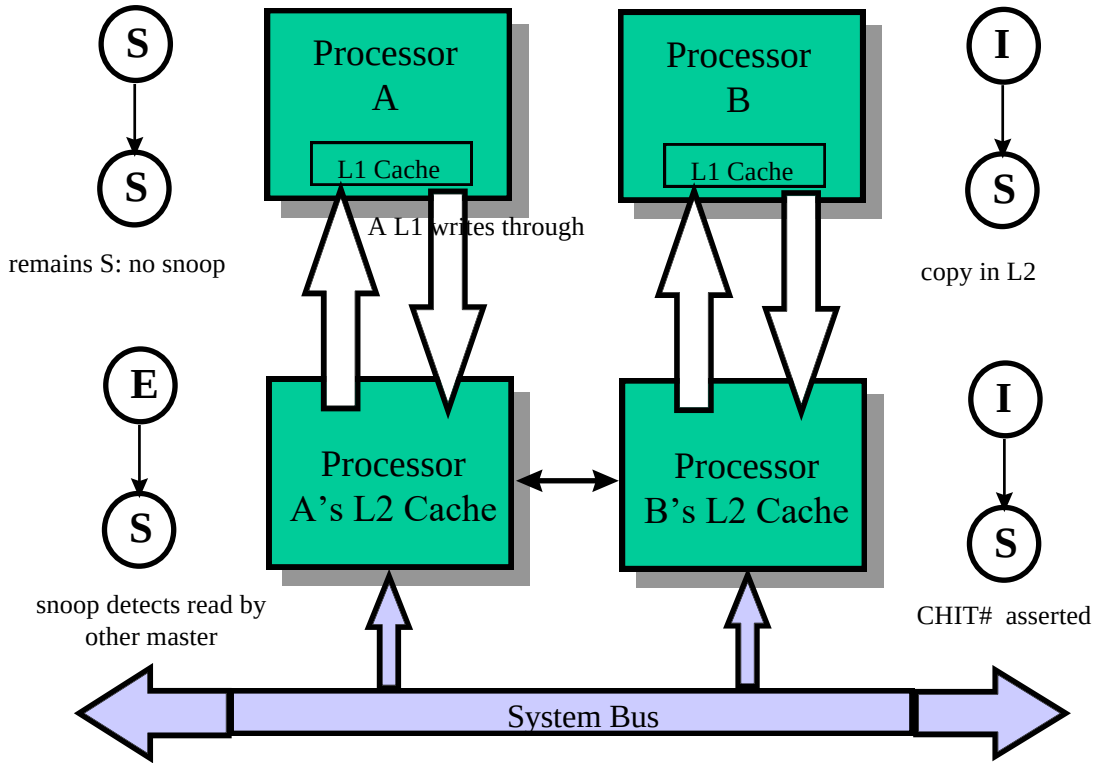
Scenario	Processor A Status	A L1 State	A L2 State
4	A holds a memory paragraph but has not modified it. Then B writes to that same line.	S	E
5	A has read a line then writes the same line once, L1 using *write through.* Then B writes to that same line.	E	M
6	A has written some line more than once, L1 uses *write back* write policy after first write. Then B writes to same address.	M	M
7	A and B have read the same paragraph, then B writes to that same line.	S	S

Scenario 1: A reads line, then B reads same line

Initial state and actions taken:

- **A** has read a paragraph, placed it into a line, but not modified it
- **A**'s L1 is in **S** state
- **A**'s L2 is in **E** state, i.e. no other processor has copy, yet its own L1 cache does have a copy; but that is transparent to other processors
- See figure: *"Initial **Read** of cache line into ..."*
- **B** has not read any data at all, thus **B**'s L1 and L2 are both in **I**
- **B** next intends to read the same line

A Reads Line, **then B Reads** Same Line

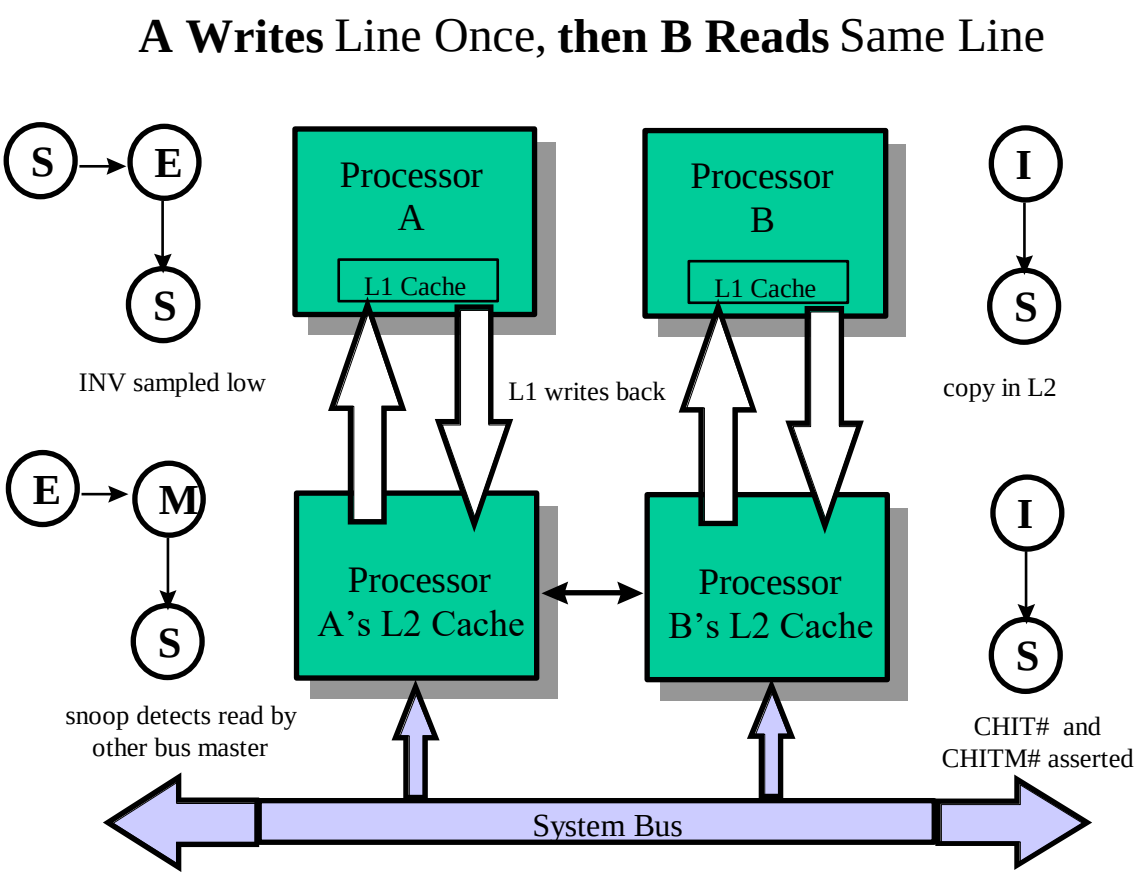


- **A**'s L2 snoops and detects a *snoop hit on read*
- **A**'s L2 does not request back-off, since request by other bus master is for *read* and its own state is **E**, not **M**
- **A**'s L1 is in **S** state; according to *write once* policy: it stays **S**
- **A**'s L2 transitions from **E** to **S**. It is aware another copy exists soon, in processor **B**
- The whole line is streamed into **B**'s L2 and then into L1 cache
- **B**'s L1 transitions from **I** to **S**
- **B**'s L2 transitions from **I** to **S**; since **A** holds a copy of that the same paragraph, **B**'s L2 state cannot be **E**
- 4 lines have copies of the memory paragraph, none are *modified*

Scenario 2: A writes line once, then B reads same line

Initial state and actions taken, snarfing is NOT yet used here:

- **A** has read a line, but not modified it
- **A**'s L1 is in **S** and L2 in **E** state, since no other processor's cache has a copy
- **B** has not read any data at all, thus **B**'s L1 and L2 are in **I**
- **A** now writes the line; experiencing a write-hit!
- **A**'s L1 transitions to **E**, switches to *write back* due to *write once*
- **A**'s L2 transitions to from **E** to **M**; new data are in L2, not in memory
- **B** now intends to read the same line, is still in **I** state



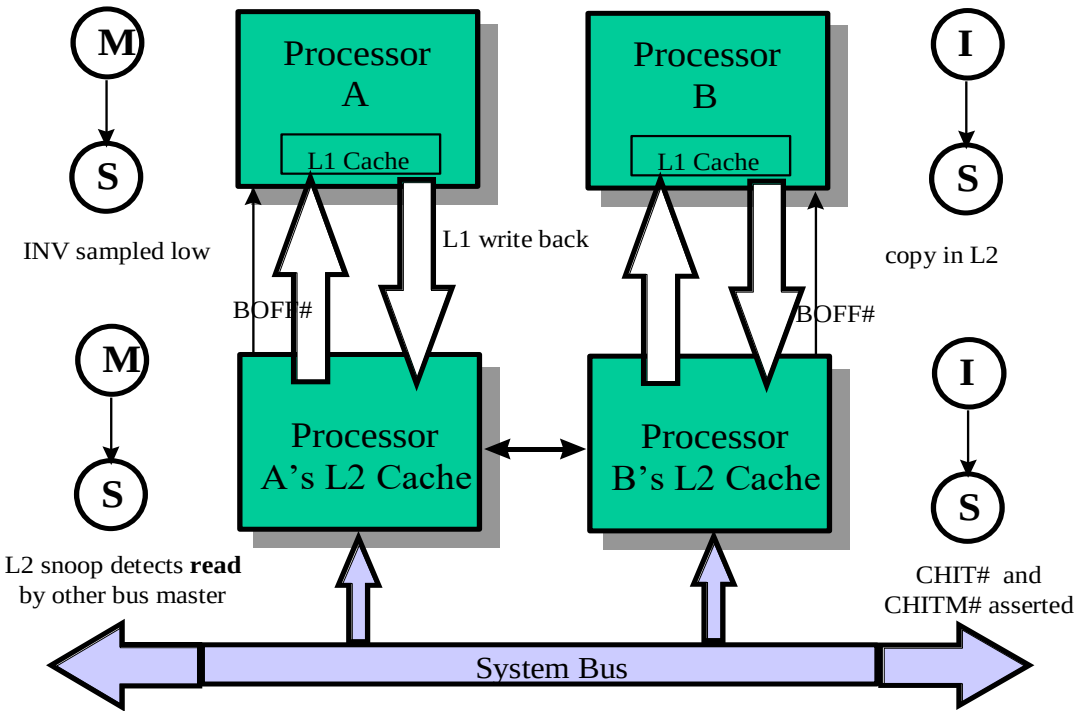
- **B**'s L1 and L2 experience read miss, L2 sends read request to bus
- **A**'s L2 snoops, sees a read snoop hit, and forces **B** to back-off
- **A**'s L1 notices due to the **E** state that it already has the newest data, and not subsequently modified them
- **A**'s L1 transitions from **E** to **S**, since the L2 cache already has a copy
- **A**'s L2 writes back data to memory, transitions from **M** to **S**, releases *back-off*
- **B**'s L2 streams in the whole line, transitions from **I** to **S**
- **B**'s L1 gets copy of line, transitions from **I** to **S**

Scenario 3: A reads, writes line multiple times, then B reads line

Initial state and actions taken, snarfing NOT used here, assuming write-by:

- **A**'s L1 has read, then written a line using *write through*, transitions to **E**
- **A**'s L2 transitions to **M**; note that the new data are in L2, not in memory
- **A**'s L1 changes to *write back* due to *write once* policy
- **A** performs a write again, hits L1 cache
- **A**'s L1 cache transitions from **E** to **M**, **A**'s L2 cache remains in **M** mode
- The modified data did not get copied to memory, as L2 uses *write back;* now **3 different copies of the same paragraph** exist!!!
- **B** intends to read the same line, is in **I** state; what happens with caches?

A Writes Line Multiple Times, **then B Reads** Same Line



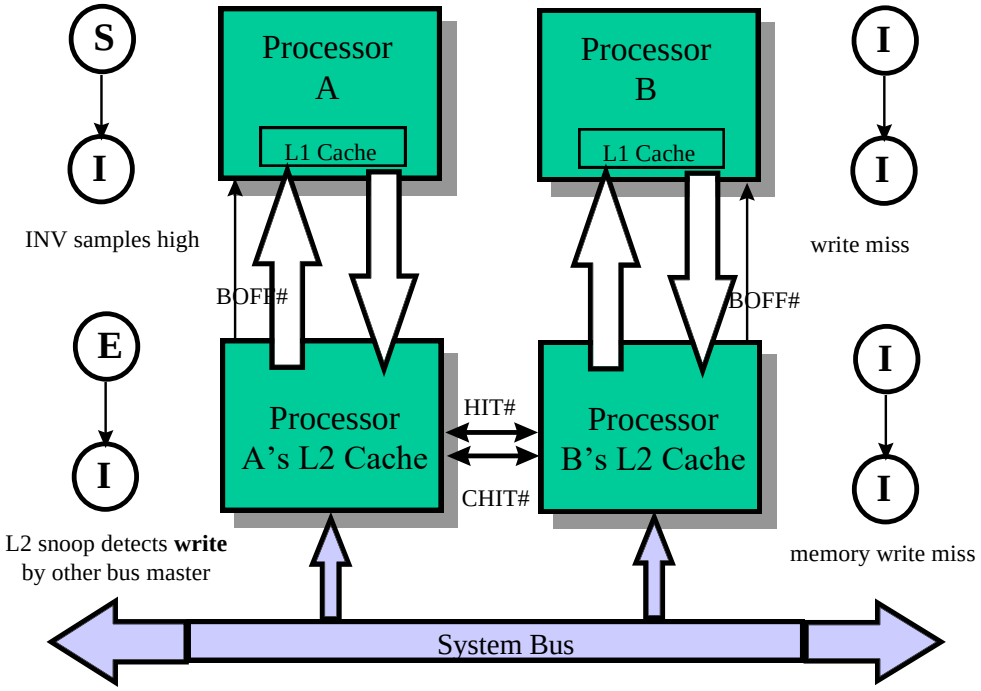
- **B**'s L1 and L2 experience read miss, **B** sends read request to system bus
- **A**'s L2 snoops, experiences a read snoop hit, realizes that **B** would get stale data, and forces **B** to back-off
- **A**'s L1 has the newest data, visible through **M** state
- **A**'s L1 writes the modified data back into L2
- **A**'s L2 writes data back to memory
- **A** and memory are in synch; **instead of 3 copies, now there exists 1**!
- **A**'s L1 state transitions from **M** to **S**, L2 from **M** to **S**, releases back-off
- **B**'s L2 streams in the whole line, second try; transitioning from **I** to **S**
- **B**'s L1 gets copy of line, transitions from **I** to **S**; all 4 are in state **S**

Scenario 4: B writes line also present and unmodified in A

Initial state and actions taken, snarfing not yet used here, assume write-by:

- **A** has read a line from memory, but not modified it
- **A**'s L1 is in **S** state
- **A**'s L2 is in **E** mode, as L2 believes no other processor has copy
- **B** has not read any data at all, thus **B**'s L1 and L2 are both in **I**
- **B** next intends to write to that same line; note use of write-by!

B Writes Line of Which **A Has Copy**



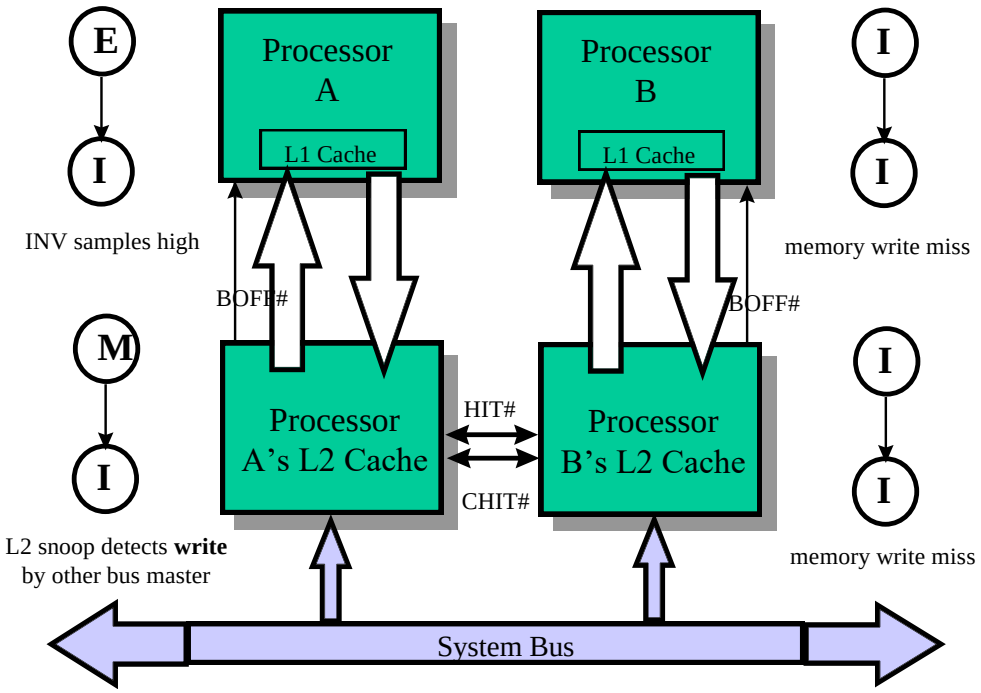
- **B**'s L1 and L2 experiences data cache write miss, L2 initiates memory write bus cycle on system bus to update memory; we use *write-by*, not *allocate-on-write*
- **B**'s L1 remains in **I**, L2 similarly stays in **I**
- **A**'s L2 detects memory write bus cycle, snoops the address, which hits
- **A**'s L2 is in **E** state, which says that a write in **B** is about to update memory of which **A** has an exclusive copy; but **A** no longer has a valid copy, thus is transitions to **I**
- **A**'s L1 also transitions to **I**
- Note that Pentium does not use *allocate-on-write*, hence all lines are **I**
- Snarfing could be used for a way better policy; see Pentium Pro

Scenario 5: A reads + writes line, then B writes same line

Initial state and actions taken, snarfing not used, assume write-by:

- **A** has read a line from memory, L1 is **S** and L2 in **E** state
- **A** writes line: **A**'s L1 writes through, updates L2
- **A**'s L1 transitions to **E**, switches policy to *write back,* L2 transitions to **M**
- **B** has not read any data at all, thus **B**'s L1 and L2 are both in **I**
- **B** next intends to write to that same line, using write-by

A Reads + Writes Line then **B Writes** Same Line

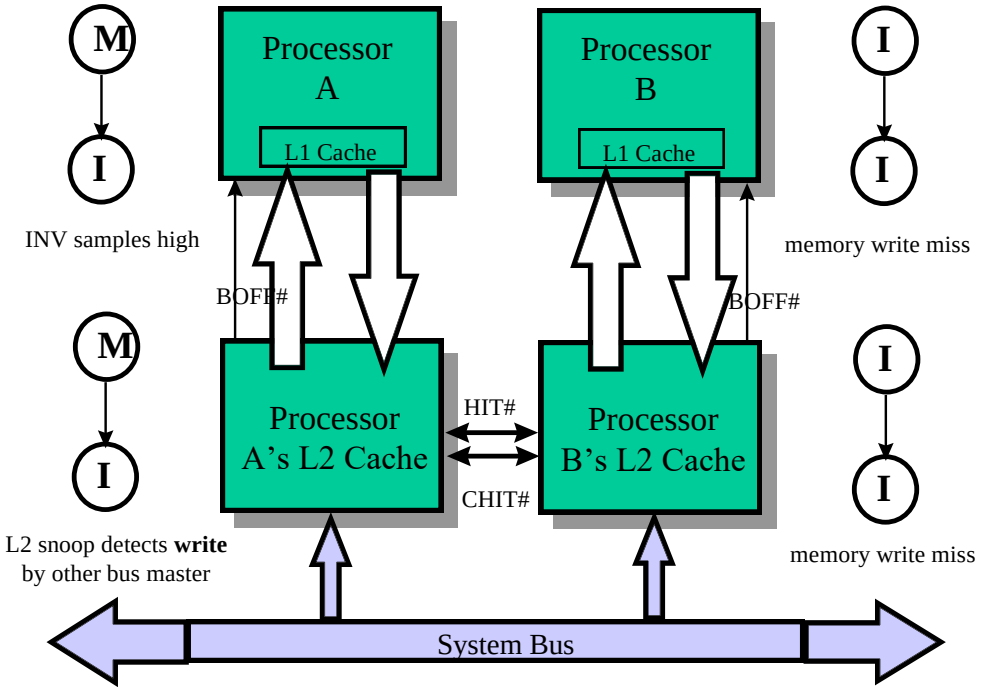


- **B**'s L1 and L2 experience cache write miss, initiate write to memory
- **B**'s L1 transitions remains in **I**, L2 similarly
- **A**'s L2 detects memory write bus cycle, snoops address, which matches
- **A**'s L2 is in **M** state, which says that write in **B** would create stale memory
- **A**'s L2 causes **B** to back-off from writing, checks if L1 made further writes
- **A**'s L1 is not in **M**; L2 writes back data to memory
- **A**'s L2 transitions from **M** to **I**, knowing that **B** will write data
- **A**'s L2 releases back-off, forces L1 to transition from **E** to **I** state as well
- **B** completes write-by, does not fill cache line, L1 and L2 end up in **I**
- Note: If **B** would use "allocate-on-write", it could *snarf* while **A** writes-back, then modify the *snarfed* line, mark it as **M**, and A would end up in **I**

Scenario 6: A reads then writes line repeatedly, B writes same without read

- **A** has written a line repeatedly, switched from *write through* to *write back*
- **A**'s L1 is in M state, and L2 is in M mode as well
- **B** next intends to write to that same line

A Writes Line Repeatedly then **B Writes** Same Line

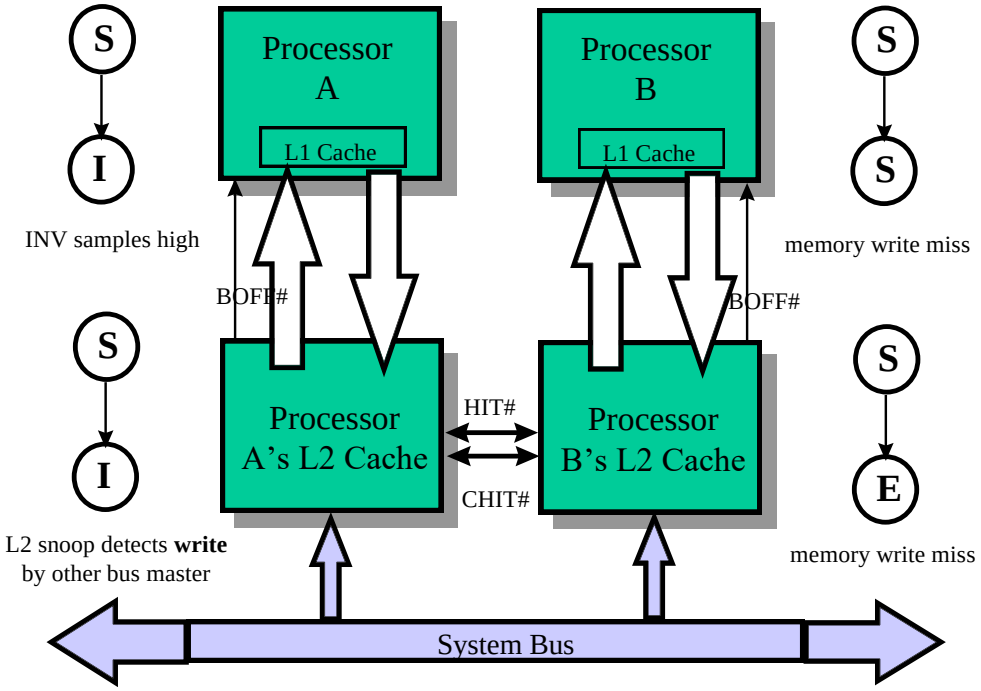


- **B**'s L1 experiences write miss, initiates write
- **B**'s L2 similarly experiences cache write miss, initiates write on bus
- **B**'s L1 and L2 is invalid I
- **A**'s L2 detects write bus cycle initiated by **B**, snooped address matches
- **A**'s L2 is in M state, which says that a write by **B** would access stale mem.
- **A**'s L2 causes **B** to back-off from writing, checks if L1 made further writes
- **A**'s L1 is in M state, thus has newer data than L2
- **A**'s L1 writes back data to L2, transitions to I, L2 writes to memory
- **A**'s L2 transitions to I, knowing that **B** will write data. Note that memory is now NOT *stale* after the current write by **A**, and **B** has NOT yet written
- **A**'s L2 releases back-off, so **B** can resume (retry) the write to memory
- **B** completes write according to write-by, still has no line in cache, **B**'s L1 and L2 end up in **I**
- Note in some other protocol **B** could *snarf* the line, use allocate-on-write. After *snarfing*, **B** could modify line, transition to **M**, leave **A** in **I**, and memory would be safe due to **B's M** state

Scenario 7: A and B have read the same line, then B writes to line

- **A** and B have read the same paragraph
- **A**'s L1 is in state S, and L2 is in **S** mode as well, memory is up to date
- **B**'s L1 and L2 are in **S**, next **B** intends to write to that same line

A and B read same line, **then B writes** to that line



- **B**'s L2 experiences write hit, and initiates memory write, because line is **S**; note that L1 would not write through, if it were in state **E**
- **B**'s L2 transitions to **E**, knowing other snooping caches will transition to **I**
- **B**'s L2 actually writes, though it generally uses write-back; but it was in **S**; write-back is only used in state **E**
- **B**'s L1 transitions to S, so that L2 would "know" of subsequent writes
- **A**'s L2 snoops the address, sees the write by **B**, transitions to **I**
- **A**'s L2 instructs L1 to snoop, which also hits and causes transition to **I**
- Since neither **A**'s L1 or L2 have modified the line, the write in **B** can proceed
- The states are: **A**'s L1 and L2 are in **I**, and **B**'s L2 is in **E** and L1 in **S**
- You probably expected **B**'s write to be held-back and **B** to end up in state **E** for L1 and **M** for L2; but the write does take place in the MESI protocol

Differences in Pentium Pro processor L2 cache management

See ref [3] for detail.

- Pentium ® Xeon ™ is designed for 4-processor MP configuration
- L2 cache is in separate cavity on Pentium Pro, but on same chip, wire-bonded
- L1 and L2 cache snoop simultaneously, hence L1 and L2 can be in E state simultaneously!!
- L1 cache on Pentium Pro (before Klamath) is twice the size, 8 KB each, code and data cache
- L2 in Pentium Pro performs *snarfing*
- On Pentium Pro, L2 unified cache is 4-way set-associative, the L1 data cache is 2-way, and the instruction cache 4-way set-associative
- Streaming into cache uses *toggle-mode*, or *critical-quad-first* mode. This resolves the hit, having caused the miss, in the shortest possible time
- Pentium Pro has no instruction delimiter bit per instruction byte in the I-cache
- Pentium Pro *squashes*

Table Showing Toggle Mode in Pentium Pro, Critical-Quad-First Mode

Address	First Quad	Second Quad	Third Quad	Fourth Quad
0x0..0x7	0x0	0x8	0x10	0x18
0x8..0xf	0x8	0x0	0x18	0x10
0x10..0x17	0x10	0x18	0x0	0x8
0x18..0x1f	0x18	0x10	0x8	0x0

Bibliography

1. Don Anderson and Shanley, T., MindShare [1995]. *Pentium ™ Processor System Architecture*, Addison-Wesley Publishing Company, Reading MA, PC System Architecture Series. ISBN 0-201-40992-5.
2. Pentium Pro Developer's Manual, Volume 1: *Specifications*, 1996, one of a set of 3 volumes.
3. Pentium Pro Developer's Manual, Volume 2: *Programmer's Reference Manual*, Intel document, 1996, one of a set of 3 volumes.
4. Pentium Pro Developer's Manual, Volume 3: *Operating Systems Writer's Manual*, Intel document, 1996, one of a set of 3 volumes.
5. Y. Sheffer: http://webee.technion.ac.il/courses/044800/lectures/MESI.pdf